Part 1 Industry Trend

ユーティリティ・モデルの現状
理論と現実の摺り合わせに関するさまざまな解

コンピュータの低価格化とネットワーク化が進行した結果、業務の処理量に合わせて複数台のコンピュータを組み合わせて利用することがごく普通に行なわれるようになった。メインフレーム時代のシンプルなモデルは過去のものとなった。しかし、現在では増えすぎたサーバの台数が管理限界を超える、という問題が生じ、その複雑さが非効率と見なされ始めた。そこでユーティリティ・モデルが脚光を浴びることとなったが、基本的な概念は単純でも、実現手法は各社まちまちとなっている。

ITシステムの現状

現在のIT環境では、ネットワークの利用が前提となっている。さらに、業務処理を複数のより小さい単位に分割し、要素ごとに異なるコンピュータで実行し、結果を繋ぎ合わせて1つの大きな業務処理とする、という手法が一般的になってきている。こうした分散処理は、どのような形で処理を分割し、どう繋ぎ合わせるのか、という点でさまざまな考え方があり得るのだが、現在一般的になってきているのは、3階層モデル（3 Tier Model）と呼ばれるアーキテクチャだ（**図1**参照）。

3階層モデルでは、業務処理（アプリケーション）をプレゼンテーション、ロジック、データ・ストレージの3層に分割し、それぞれに対応するサーバを配置してネットワークで接続する。このモデルの背景には、インターネットの普及、そしてインターネット環境で成立した技術を社内の業務ネットワークで利用するというイントラネットの考え方がある。

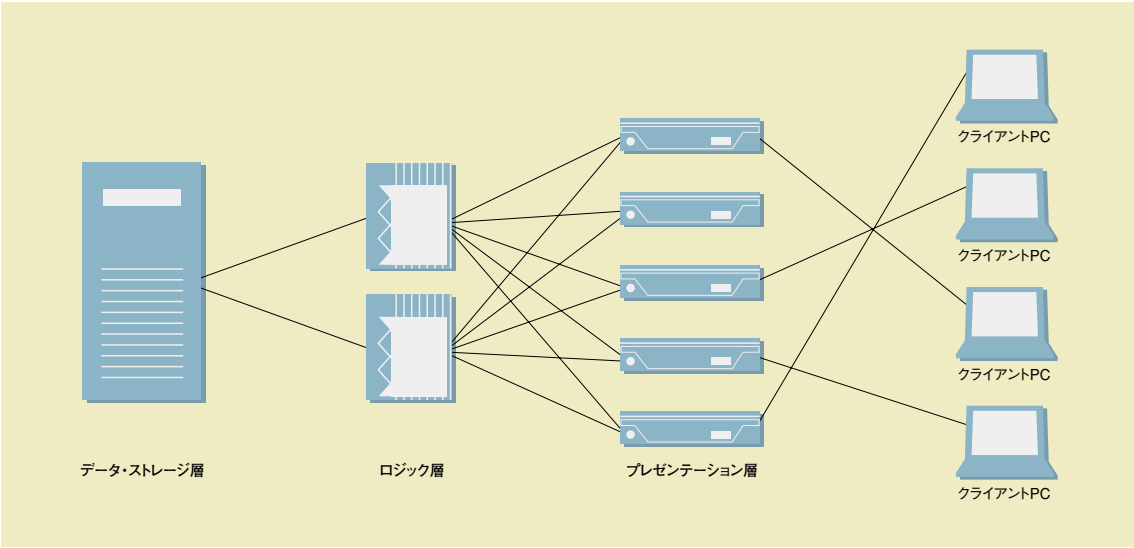

図1：3階層システムの基本的な構造
クライアントから殺到する大量のリクエストを効率よくさばくため、各層でそれぞれ処理の一部を分担しながら処理を集約していく、というのが基本的な考え方だ。

3階層モデルを実際のサーバの処理に注目して見てみると、まずアプリケーション層を構成するのはWebサーバとなる。Webサーバは、クライアントPC上で実行されるWebブラウザからのリクエストに応じてWebページを送出する。このWebページがユーザーにとってのグラフィカル・ユーザー・インタフェースとなる。具体的には、多数の入力フィールドを備えたフォームや帳票といったビジュアル・イメージになるようにあらかじめ準備されたWebページであり、これがアプリケーションの操作画面そのものとなる。より詳細には、プレゼンテーション層はサーバ・サイドのWebサーバとクライアント・サイドのWebブラウザの組み合わせで構成されており、その間はTCP/IPネットワークの技術と、Webのためのアプリケーション・プロトコル(HTTP)で接続される。

Webページは、もともとはハイパーリンクを埋め込まれた静的なHTMLドキュメントだったが、すぐに拡張が行なわれ、外部のプログラムを使って動的にHTMLファイルを生成することができるようになった。フォーム・ページでユーザーからの入力を受け取ってWebサーバに戻し、Webサーバがこの情報を外部のプログラムに伝えることで、ユーザー入力に対応する内容を表示するWebページ(HTMLファイル)を自動生成することができる。これで、ユーザーに情報を表示し、ユーザーの入力に応じて表示を更新する、という基本的なユーザー・インタフェースのI/O処理が実現できる。

と同時に、動的なページ生成のために呼び出される外部プログラムは、単にHTMLファイルを書き出すだけに留まらず、さらに他の業務処理プログラムと連携して業務処理の中核部分を実行することになる。これが、ロジック層に置かれるアプリケーション・サーバの役割となる。プログラミング・モデルによってさまざまな技術要素が持ち込まれ、もっとも複雑化している層でもある。Webサーバとの通信は、CGIやネイティブAPI、JSPやJava Servletの利用など、洗練と標準化が進行する過程で徐々に移り変わりつつあるが、一貫したテーマとなっているのは、ビジュアル・デザインとアプリケーション・ロジックの明確な分離である。

最後に、データ・ストレージ層にはRDBMSが置かれるのが一般的だ。アプリケーションの処理結果や初期データ、そしてユーザーが入力したデータを保存する。アプリケーション層とのインタフェースには従来のアプリケーションと同様のSQLやODBC／JDBCといったインタフェース／プロトコルが利用される。

各層では、処理の内容の違いに対応してサーバに求められる要件も変わってくる(**図2**参照)。プレゼンテーション層のWebサーバは、クライアントからの

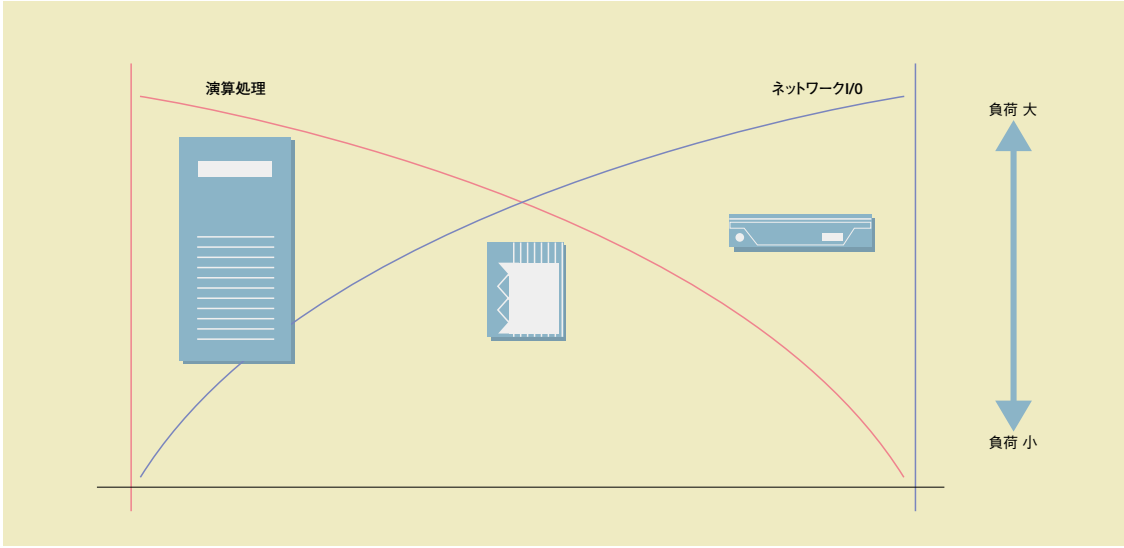

図2：各層のサーバの性格の違い
各層のサーバは、担当する処理の違いから要求性能も変わってくる。データ・ストレージ層のサーバは演算処理やファイルI/Oの性能が求められ、プレゼンテーション層のサーバにはさほどの演算処理能力はいらない。

処理要求の受付窓口の役割を果たすため、大量のネットワーク・リクエストを効率よくさばくために通信帯域の確保が欠かせない。一方で、Webサーバ・ソフトウェアを実行してコンテンツの配信を行なうのが主な役割であることから、ネットワークとのデータの送受信が処理の中心であり、負荷の高い演算などはあまり行なわれない。また、HTTPの特性上、各リクエストは基本的にステートレスで行なわれる。つまり、あるユーザーの処理の途中経過をすべて把握している必要はなく、一回のリクエストに一回応答すればそれで終わり、というのが基本となる。1画面のWebページには多数の構成要素があり、HTMLファイルに加えて画像データなども多数含まれるが、基本的な処理はファイル単位で行なわれ、一回の処理はリクエストに応じて1つのファイルを送出すれば終わる。必要なファイルをすべてリクエストし、集まったファイル群から1画面のWebページを構成して表示するのはクライアント側のWebブラウザの仕事である。こうした特性から、Webサーバを実行するサーバマシンは、少数の高性能マシンよりも、比較的低性能な安価なマシンを多数並列に配置しておく方が全体の処理能力(この場合はネットワーク・スループット)を高めることが容易で、コスト面でも安価になる。

対照的なのがデータ・ストレージ層のサーバで、データベースを運用し、データの一貫性を保つためにはデータベース・インスタンスは1つである方が好ましい。安全性/信頼性の面からクラスタ構成を採ることが多いが、それでも論理的には1台の巨大高性能サーバを配置することが一般的だ。

この中間的な性格をもつのが、ロジック層のアプリケーション・サーバだ。アプリケーションでは処理の一貫性や順序正しい逐次実行などが求められるので、ステートレスなWebサーバの背後でアプリケーションの実行状態の保持に責任を持つ。また、アプリケーション・ロジックが実行されることから、アプリケーションの処理内容によっては演算性能も相応に求められることになる。万一のサーバダウンでも実行中の処理が失われることがないよう、クラスタリングを行なうことも一般的だ。そこで、一般的にはミッドレンジサーバを数台〜数十台配置することになる。

このように、3階層モデルは1つのアプリケーションを3階層に分割し、それぞれに異なる性格付けを行ない、異なるハードウェア上で実行するモデルである。3階層モデルはイントラネット・アーキテクチャとしては標準的なもので、現在稼働中のWebアプリケーションのほとんどがこのアーキテクチャに沿って作られているといってよいだろう。ここで問題は、3階層という把握しやすい数の階層に整理されたとはいえ、実際のサーバの構成はかなり複雑で、特にWebサーバ層ではサーバ台数が膨大になる点だ。

管理限界超過/利用効率低下

サーバの台数が増えると、管理の手間は大幅に増えることになる。しかも、前述の通り、3階層モデルでは各層ごとに性格の異なるサーバを配置している。コスト面での最適化を考えて、Web層では安価なエントリ・サーバを多数配置する。数が多いことから、1Uサーバや、最近ではブレード・サーバなどが活用される。ロジック層では、多数のWebサーバからのリクエストに応じてアプリケーションを実行するため、4CPU以上くらいの構成のミッドレンジ・サーバを使う例が多い。1Uのエントリ・サーバとは価格が桁違いなので、数的にはぐっと少なく、処理内容によっては2台ということも珍しくはない。一方、最終的にすべてのデータ・アクセスが集中することになるデータ・ストレージ層のサーバはハイエンド、もしくはミッドレンジのサーバで、処理の規模に応じて4〜8CPU、場合によっては16CPU以上の大規模なサーバを使用する。

このとき、どのくらいの処理能力をもつサーバを用意するのが最良なのか、その判断は極めて困難なものになる。クライアントのリクエストを受け付けるWebサーバの処理能力は、クライアント数やそれぞれの処理の頻度、HTMLファイルのサイズや画面数に依存する。そして、Webサーバの数が決まったとして、その後段に位置するロジック層のアプリケーション・サーバはどのくらいの処理能力のサーバを何台用意すれば適切なのか。これを適切に予測することは実際にはかなり困難で、現場ではサンプル・アプリケーションまたは本番用アプリケーションをあらかじめ用意し、想定される業務負荷を実際に掛けてみてパフォーマンスを測定し、その結果に基づいて判断する、という手法が採られる。つまり、やってみなくてはわからない、というのが現実なのである。当然、データ・ストレージ層のRDBMSサーバも同様だ。

このとき、業務を安定して実行できることを重視すれば、各サーバの処理能

力には一定の余裕を持たせておくことになる。万一の負荷集中に耐え、システムダウンなどで業務が停止してしまうことを避けるためには当然の配慮である。しかし、この配慮もそもそも業務で必要となる処理能力を正確に見積もることが困難であることから、余裕を大きく取りがちである。実際にシステム構築を担当する立場からは、万一処理能力不足でサーバがダウンしたりすれば大変な責任問題に発展する可能性が高いため、安全策としては「絶対不足しないだけの処理能力をあらかじめ確保しておく」しか対処法がないともいえる。

景気がよく企業が右肩上がりで成長を続け、業務処理量も増加の一途を辿り、やがてサーバの処理能力が不足して増強を余儀なくされる、という時代には、明らかに余剰な処理能力を持たせることが問題視されることはなかった。構築時点では余剰でも、いずれは消費され尽くされるのであれば無駄とは言えない。むしろ、成長に合わせて頻繁に増強を繰り返すよりは、ある程度まとめて処理能力を確保してしまい、システム改修の回数を減らす方が合理的だという考え方も成立するだろう。しかし、インターネット・バブルと呼ばれたこうした急成長期は既に過ぎた。ユーザー企業は運用コストの削減に真剣に取り組むようになり、明らかに余分な処理能力を保有し続けることの無駄がクローズアップされてきたのである。業務処理も、常に一定量の処理を行なうのであれば適切な処理能力の見積もりも容易になるが、実際には日次／月次、あるいは時間単位で変動があるのが普通だろう。会計システムでは決算期に処理量が急増するだろうし、ユーザー向けのオンライン販売システムでは賞与時期や贈答需要が発生する時期に取扱量が急増し、閑散期には激減するといった季節変動があるだろう。こうした変動に対し、これまでは「予想される最大需要を満たすだけの処理能力に、さらに安全弁としての余剰を加えたものを適正量とする」という見積もりを行なってきたわけだ。

しかし、業務処理量の変動幅によっては、この方針は無視できない量の遊休資産を生み出すことにもなる。たとえば米国では、個人消費の大半がサンクスギビングからクリスマスの、いわゆるホリデーシーズンに行なわれるといわれている。オンライン販売サイトなどでは、この時期の取扱量は平常時の10倍以上に達するのも珍しくはないという。このとき、システムの処理能力を最大需要に合わせて見積もったとすると、一年の大半は、その90%が無駄に遊んでい

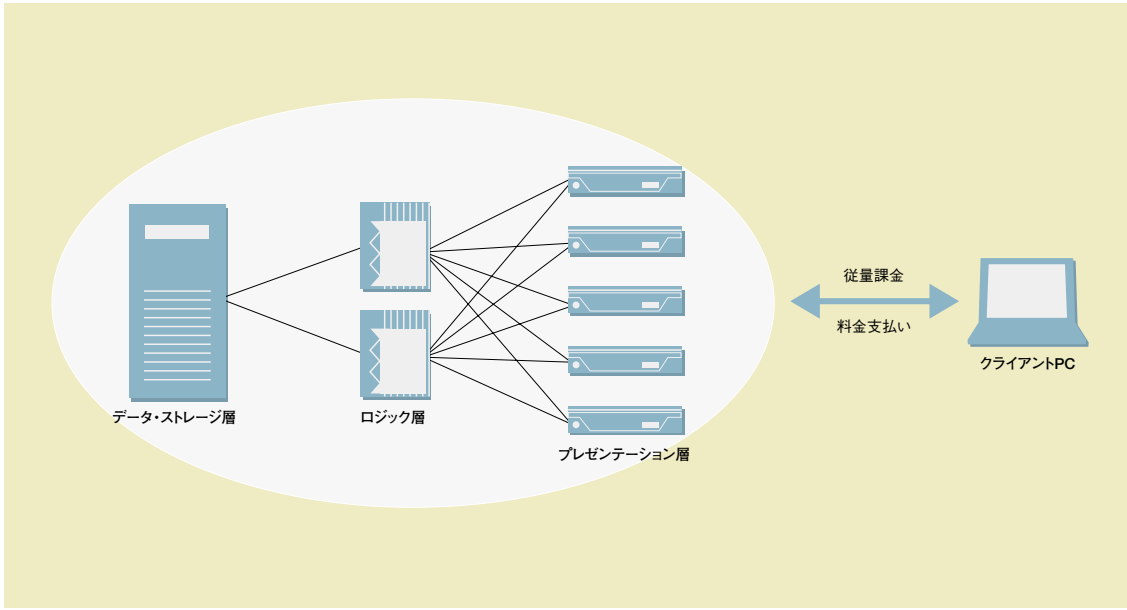

図3：ユーティリティ・モデルの概念
図1に示すような複雑な3階層システムの詳細をサービス提供者が隠蔽し、ユーザーに「使用量に基づく従量課金」という単純なインタフェースで提供することがユーティリティ・モデルの基本的な考え方である

ることになるわけだ。

3階層システムは、役割の異なる3層のサーバを相互接続して実現する、それ自体が複雑さを内包したシステム・アーキテクチャである。処理能力を増強するためには各層ごとのサーバの台数を増やしていくことになるため、相互接続の組み合わせの増大から、複雑さはさらに増していく。こうした複雑なシステムを安定的に運用管理していくためには専門知識を備えたエンジニアが必須となり、多額のコストがかかる。さらに、それだけのコストを掛けて維持しているシステムの大半が無駄に遊んでいるとなれば、トータル・コストのレベルでは極めて投資効果の低いものと判断せざるを得なくなる。これが、現在企業が直面しているIT投資の最大の問題であろう。

ユーティリティ・モデルとは何か

システム構成自体が複雑で運用管理コストが増大することと、保有しているシステムの利用効率が低く、結果的に過大な投資が行なわれていること。この2点をユーザー企業が問題視するようになってきたことに対し、ITシステム・ベンダーから提示された解の1つが、ユーティリティ・モデルである。厳密な定義ではないが、本誌でも何度も取り上げているユーティリティ・コンピューティングが理論的なアーキテクチャとすれば、ユーティリティ・モデルはそのアーキテクチャをユーザーが利用できる形にビジネス化したものと捉えることができるだろう(**図3**参照)。

ユーティリティ・モデルをもっとも簡単に実現するなら、ITシステムをアウトソーシングし、利用量に応じて料金を支払うという形がもっとも自然な実装となるだろう。実際、こうした形でのユーティリティ・モデルの提供に踏み切る企業も順次増えてきている。

技術的な課題としては、「ユーザーの使用量」をどう測定し、「システム使用料」をどのように算出するのがよいのか、という問題がある。

歴史的には、ユーティリティ・モデルはメインフレームの時代にはごく当たり前に実現されていたことだ。というのも、メインフレームは巨大で高価なリソースであり、1社だけの処理でそのすべてを使い切るとは限らなかったためだ。そこで、データセンターに設置されたメインフレームのリソースをパーティショニングで分割したり、あるいはTSS(Time Sharing System)で共有利用したりするという使い方が普通に行なわれていた。

メインフレームを設置し、サービスを提供する事業者は、複数の顧客を集め、総計でちょうどメインフレームの処理能力を使い切るように配置できれば、手持ちの資産を効率よく活用してビジネスを展開することができたわけだ。

メインフレームの時代には、あらゆる処理が1台のコンピュータの内部で実行されていたため、その使用量を測定する手段もさまざま考えられた。CPUのクロックサイクル、メモリやストレージの使用量の積算値、I/O処理の回数やプリントアウトの枚数など、さまざまな指標で計測された使用量に処理ごとの単価を掛けて合計することで、ユーザーが支払うべき使用料を算出することができた。しかも、こうした統計値にもとづく課金はユーザー側から見ても合理的なものと考えられたため、提供者/利用者双方が受け入れやすい体系でもあった。

一方、3階層システムにもとづく分散システムの場合、システムの使用量を計測するのは、それだけで一仕事である。そもそも、処理能力も価格も大きく異なる複数のコンピュータを組み合わせてネットワーク接続してシステムを構築しているので、どの部分で何を測定すればユーザーのシステム使用量を把握できるのか、明らかではない。また、メインフレームのように複数のユーザーがシステムを共有する前提で設計されているわけではないUNIXサーバの場合、複数の顧客のシステム、あるいは性格の異なる複数のアプリケーションを1台のサーバに同居させることは極力避けるのが一般的である。このため、使用量を正確に測定できたとしても、使われなかった余剰リソースを別の用途に転用できないことも多い。この場合、提供者としては使用分だけ請求すべきなのか、それとも占有分として全体を請求するべきなのか悩ましいところだろう。たとえるなら、バス運賃の決め方にも似ている。乗り合いバスなら乗車人数にかかわらず1人分の料金を支払うが、1人で1台のバスを貸し切りにすれば他の空席分の料金も支払うことになるだろう。

現時点でのユーティリティ・モデルの問題は、この使用量測定と料金算定の手法の標準がまだ確立されていない点だ。ユーザー側としても、損得の分岐点がまだ正確には把握できていない。料金設定と使用状況のバランスによるが、使用量に応じて課金されるという従量制課金体系は、使用量が増えると

支払いも高額になる。ある程度以上の量をコンスタントに使用するのであれば、むしろ従来通りシステムを自社保有する方が安上がり、ということも起こりうるが、この分岐点となる使用量を正確に把握することは困難だ。

なお、ユーティリティ・モデルの本質は、「ユーザーが納得できる算定基準に基づく従量制課金」という点にある。つまり、システム構成の複雑さや、余剰の処理能力に対する無駄な投資を回避したい、という要求に応えることが第一義である。

技術的には、複数のコンピュータをまとめて仮想的な1台の巨大コンピュータとして利用可能とするグリッドの技術が成熟してきたことによって、ある程度の顧客を確保できる事業者にとってはこうした従量制課金体系でサービス提供を行なうことができるようになってきつつある。こうして、システムの複雑さと余剰リソースの再活用という問題をユーザーから切り離して事業者の側に解決を任せるのが、ユーティリティ・モデルの本質だと言うこともできよう。

とすれば、システムの利用量をコンピュータのハードウェア／ソフトウェアに着目して測定することは必ずしも必須とは言えない、という考え方もできる。使用量について提供者／事業者の双方が納得して合意できる体系を採用できれば、測定に拘る必要はないわけだ。典型的な例が、SLAに基づいて従業員数やシート数といった単位で課金を行なうソフトウェア・ライセンスやASP（Application Service Provider）のモデルがある。これらは別段目新しい存在ではないが、ユーザー・メリットとしてはユーティリティ・モデルと同様に効果的なものだと見ることができるだろう。

現時点で完璧なユーティリティ・モデルというものはないため、ユーザーがそれぞれの処理内容や業務負荷に応じて最善解を模索する必要がある状況だ。しかし、ビジネスという視点ではユーザーの需要に応えるべくさまざまな料金体系の提案が行なわれているので、技術的な要素に固執するのではなく、ビジネス面での得失とのバランスを考えながらどのようなモデルを採用するかを判断する必要があるだろう。

ITビジネスの停滞とユーティリティ・モデル

ユーティリティ・モデルが注目される背景には、右肩上がりの成長を続ける過程でユーザーを置き去りにしてきた感のあるIT業界に対する批判の意味合いもあるように思われる。

ハードウェアはムーアの法則に従って1年半～2年で倍というペースで性能を向上させてきた。一方で、製造技術の進歩と大量生産による歩留まりの向上から、価格はほぼ横ばいという状況だ。これは、あとから購入すれば同じ価格で倍も高性能なマシンが入手可能ということだが、ほんの数年前のマシンが完全に陳腐化しており、まったく資産価値がなくなっているという状況も生んだ。減価償却が終わる前に、既に現実の資産価値は失われてしまっているのだから、中古で売却して多少なりとも取り戻すということもままならない。

ソフトウェアも同様にバージョンアップを繰り返し、速いサイクルで陳腐化が起こった。インターネット時代に入ってセキュリティ・ホール対策の重要性が増す一方、旧バージョンのサポートをさっさと打ち切ってしまうベンダーの姿勢は、無用な買い換えを強制されるという意識をユーザーに植え付ける遠因になっているはずだ。

こうした状況から、ユーザー企業の多くは、ITシステムにいくら投資しても、最新のソフトウェア／ハードウェアを購入しても、ほんの数年で資産価値がなくなるのだから、まさに乾いた砂に水をまくかのように資金が吸い込まれて行くような印象を受けても不思議はない。ユーザー企業も急成長を遂げていたインターネット・バブルの頃ならいざ知らず、現状の低成長期にあってはこうした底なしの投資は許されなくなってきている。

そこで、もうITシステムを自社で保有したくはない、利用料金を支払う代わりに常に最新のシステムを使わせてほしい、と考えるユーザーが出てくることはごく当然の事とも考えられる。昨今のユーティリティ・モデルの充実ぶりは、こうしたユーザーの不満に遅まきながらITシステム・ベンダーが真剣に向き合うようになった結果だともいえるだろう。

ユーティリティ・モデルの実現に向けた昨今の動きは、ITシステム業界が遂に成熟期を迎え、ユーザーと共に堅実な成長を志向していく段階に移行する兆しだと捉えることができる。これまでの不満要素の代表格だった、投資効果が曖昧／運用管理コストが高すぎる、という問題が、ユーティリティ・モデルやサブスクリプション・モデルといった現実的な対策によって解消されていくことを期待したい。ユーザー企業の側にも、これまで以上に自覚的なITシステム選択が求められる。